4. Structures récursives: listes et arbres

LES LISTES ET LES ARBRES sont des structures fondamentales définies par récursion. Le cas de base représent un ensemble vide (typiquement par une référence null). La cas récursif capture l'auto-ressemblance de la structure avec au moins un membre : une liste non-vide est formée par une tête et la sous-liste de ses successeurs, et un arbre non-vide est formé par sa racine et un ensemble de sous-arbres.

L'implémentation naturelle des opérations dans une telle structure exploîte la récursion. La definition récursive se traduit aussi en une représentation spécifique : la structure s'organise par des **nœuds** dont chacun contient les données associées avec un seul élément, et aussi des liens à d'autres nœuds (références aux voisins sur la liste ou aux relations dans l'arbre). En une implémentation **endogène**, on manipule les données en concert avec les liens : le contenu est inséparable du containeur. Dans les langages orienté-objet, ce style de programmation correspond à la manipulation des objets de nœuds avec les références entre eux stockées en variables d'instance. Mais on peut également manipuler les unités de donnnées séparamment des unités de la structure, et concevoir une implémentation **exogène** (p.e., basée sur l'échange de contenu entre les cases d'un tableau).

4.1 Structures récursives

Les principes fondamentaux d'organisation de données, notamment l'arrangement séquentiel ou hiérarchique, correspondent à des structures qu'on peut définir récursivement. Les listes (Déf. 4.1) et les arbres (Déf. 4.2) sont des types agrégés «auto-semblables». Une liste non-vide est la combinaison de sa tête et la sous-liste après. Un arbre non-vide est une collection de sous-arbres joints à une racine commune.

Définition 4.1. *La **liste** est une structure récursive construite par l'application des règles suivantes.*

$$\text{liste} \to \langle \textit{nœud externe} \rangle \qquad \textit{liste vide} = \text{null}$$
$$\text{liste} \to \big(\langle \textit{noeud interne} \rangle, \text{liste}\big) \qquad (\textit{tête}, \textit{successeurs})$$

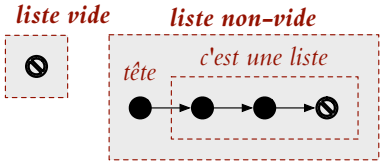

FIG. 1: Une liste est soit une référence null (un seul nœud externe), soit une paire formée par un nœud interne (la tête) et d'une autre liste.

Définition 4.2. *Un **arbre enraciné** (ou «arboresence») est une structure récursive construite selon les règles suivantes.*

$$\text{arbre} \to \langle \textit{nœud externe} \rangle \qquad \textit{arbre vide} = \text{null}$$
$$\text{arbre} \to \big(\langle \textit{noeud interne} \rangle, \underbrace{\text{arbre}, \ldots, \text{arbre}}_{d > 0 \textit{ fois}}\big) \qquad (\textit{racine}, d \textit{ enfants})$$

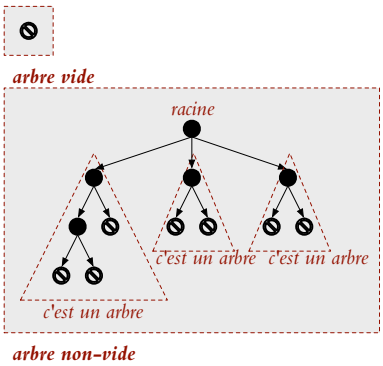

FIG. 2: Un arbre enraciné est soit null (nœud externe), ou il est composé d'un nœud interne (la racine) et un ensemble d'arbres enracinés (les enfants).

4.2 *Liste chaînée*

La structure appellée **liste chaînée**[1] (*linked list*) est une liste d'éléments conservés chacun dans un nœud qui contient aussi un ou deux liens sur le nœud suivant[2] et/ou précédent (Fig. 3). La liste est identifiée par son premier nœud, ou la **tête** (*head*). Le dernier nœud interne s'appelle la **queue** (*tail*) de la liste.

Les nœuds sur une liste chaînée standard ont deux variables : une pour stocker les données associées avec le nœud et une autre qui référence la tête de la sous-liste après (x.data et x.next). Les nœuds sur une **liste doublement chaînée** ont trois variables : une pour stocker les données associées avec le nœud et deux autres pour stocker les références aux nœuds voisins. Sur une **liste circulaire**, la queue et la tête sont liées (en place des références null des listes linéaires).

Avec les génériques[3] de Java, on peut enforcer le typage des données placées aux nœuds à compilation. Dans l'implémentation ci-dessous, la classe du nœud est parametrisé par le type des données qu'on peut y stocker (type Data).

```
/**
 * Noeud de liste avec payload typisé
 * @param <Data> type du payload
 */
class ListNode<Data>
{
   private final Data data;  // inséparables
   private ListNode<Data> next;        // prochain élément
   ListNode(Data data) // nouveau noeud sans successeur
   {

      this.data=data; this.next=null;
   }
   Data getData(){return this.data;}
   ...
}
```

Classes parametrisées. On déclare un type parametrisé[4] par `C<A, B, ...>` où `C` est la classe générique avec paramètres `A`, `B` qui sont des types (classes) eux-mêmes. Aspects importants des types parametrisés en Java .

- ⋆ Les paramètres dans la déclaration de classe appartiennent à une instance et n'existent pas dans un contexte statique.
- ⋆ Les arguments aux types prametrisés sont vérifiés lors de compilation mais l'information n'est nullement disponible à runtime. Donc,
  - ∗ on ne peut pas instancier un objet avec le paramètre-type (**new** A() ne marche pas),
  - ∗ on on ne peut pas vérifier si un objet en est membre (x **instanceof** A ne marche pas), et
  - ∗ **getClass**() ne donne aucune information sur les arguments utilisés lors de l'instanciation.

  Pour la même raison, on doit éviter des tableaux de types parametrisés

[1] W(fr):liste chaînée

[2] Notez que Déf 4.1 établit les notions de «suivant» et «précédent» avec précision, par récurrence (la tête est avant les nœuds sur la sous-liste dans le cas récursif)

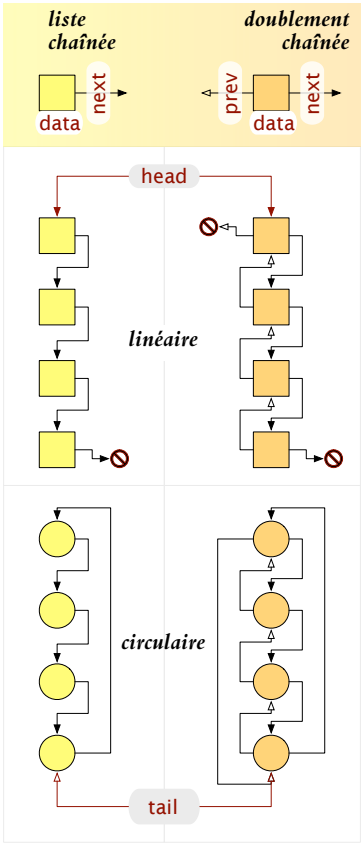

FIG. 3: Listes chaînéees avec nœuds comportant un ou deux liens. Si la liste est circularisée, on garde une référence à la *queue* : la tête est le prochain nœud.

[3] tutoriel Java : *(Java generics)*

[4] Java Language Specification : *Parametrized Types*

(comme `LinkedList<String>[10]`) car le typage ne peut pas être enforcé aux membres du tableau.

⋆ L'héritage ne se transfère pas à travers des classes génériques : même si E est une sous-classe de F, `LinkedList`$\langle E \rangle$ n'est pas une sous-classe de `LinkedList`$\langle F \rangle$.

Opérations sur la liste chaînée

La liste chaînée supporte le parcours et la manipulation locale de l'ordre. Il est simple d'insérer ou supprimer la tête ou après un nœud donné. La circularisation donne accès à la queue (et la tête). Une liste doublement chaînée est utilisée si la navigation est nécessaire aux deux sens sur la liste, p.e. lors de suppression et insertion avant un nœud car on doit changer les références chez le nœud précédent (Fig. 4).

opérations	*liste chaînée*		*liste doublement chaînée*	
	lin.	*circ.*	*lin.*	*circ.*
accéder à la tête	+			
TA pile				
accéder à la queue	-	+	-	+
TA queue				
concaténer				
prochain	+			
insérer/supprimer après				
précédent	-		+	
insérer/supprimer avant				
renverser	-		+	

FIG. 4: Opérations avec temps constant (+) sur une liste chaînée. Une liste circulaire doublement chaînée permet même l'implémentation de renversement de liste, avec l'astuce de garder un bit à côté pour interpréter `.next`/`.prev` comme prochain/précédent ou vice versa.

Insertion et suppression. Ajouter une nouvelle tête ou «couper» la tête d'une liste est triviale. Insertion ou suppression après la tête nécessite l'affectation de deux références ou une référence (Fig. 5) :

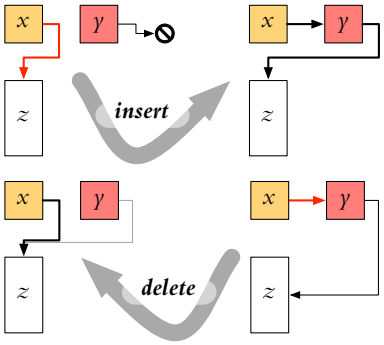

FIG. 5: Insertion et suppression après la tête.

```
class ListNode<Data>
{
   private Data data;
   private ListNode<Data> next;
   ...
   void insertNext(ListNode<Data> y){
      LisrNode<Data> z = next;
      y.next = z;
      next = y;
   }

   void deleteNext(){
      ListNode<Data> y = next;
      ListNode<Data> z = y.next;
      next = z;
   }
   ...
```

Implémentations récursives. On peut exploîter la récursivité de la structure dans l'implémentation en décomposant une opération sur une liste non-vide :

- ⋆ travail sur le nœud à la tête
- ⋆ travail sur la sous-liste après la tête [par appel récursif]
- ⋆ combinaison des résultats sur la tête et sur la liste de successeurs [mieux de délégeur vers l'appel récursif pour récursion terminale]

```
class ListNode<Data>
{
   private Data data;
   private ListNode<Data> next;
   ...
   /* accès aux noeuds par indice */
   ListNode<Data> getNode(int i)
   {
      if (i=0) return this;
      else return next.getNode(i-1);
   }
   /* recherche */
   boolean contains(Data key)
   {
       return data.equals(key)
         || (next != null && next.contains(key));
   }
   /* ... */
```

```
   /* ... */
   /* suppression par indice; retourne la nouvelle tête */
   ListNode<Data> deleteAt(int i)
   {
      if (i==0) { return node.next;}
      else {next=next.deleteAt(i-1); return this;}
   }
   /** longueur de la liste
    * @param x tête de la liste (possiblement null)
    * @param L 0 au premier appel
   */
   static int length(ListNode<?> x, int L)
   {
      if (x==null) return L;
      return length(x.next, L+1);
   }
   ...
} // class ListNode
```

Implémentations itératives. La liste accommode les implémentations itératives très bien aussi.

```
class ListNode<Data>
{
   private Data data;
   private ListNode<Data> next;
   ...
   static <D> ListNode<D> getNode(ListNode<D> head, int i)
   {
      ListNode<D> node = head;
      while (i>0) {node = node.next; --i;}
      return node;
   }
```

```
   static boolean contains(ListNode<?> head, Object key)
   {
      ListNode<?> node=head;
      while (node != null && !node.data.equals(key))
         node = node.next;
      return (node!=null);
   }
   static int length(ListNode<?> x)
   {
      int L = 0; for (; x!= null; x=x.next) L++;
      return L;
   }
```

Sentinelles

On peut utiliser une sentinelle[5] pour dénoter la tête et/ou la queue.

Définition 4.3. *Une **sentinelle** est un élément factice dans une structure de données.*

Avantage : code plus clair, exécution un peu plus rapide. Désavantage : un nœud de plus → n listes de longueur totale ℓ nécessitent $n + \ell$ nœuds au lieu de ℓ.

[5] W(en):Sentinel

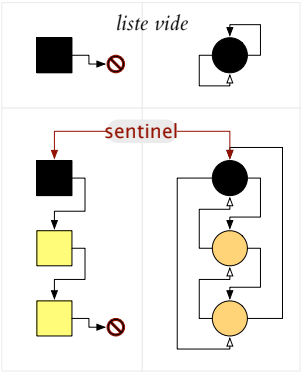

FIG. 6: Listes linéaires et circulaires avec sentinelle.

4.3 Arbres

Selon Définition 4.2, un arbre enraciné T est une structure définie sur un ensemble de nœuds qui

1. est un **nœud externe**, ou
2. est composé d'un **nœud interne** appellé la **racine** r, et un ensemble d'arbres enracinés (les **enfants**)

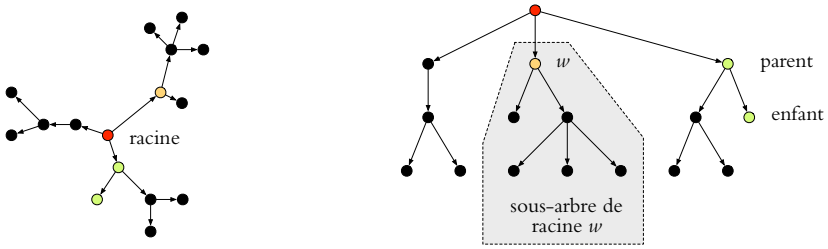

FIG. 7: Arbre enraciné.

Définition 4.4. *Un **arbre ordonné** T est une structure définie sur un ensemble de nœuds qui*

1. *est un **nœud externe**, ou*
2. *est composé d'un **nœud interne** appellé la **racine** r, et les arbres $T_0, T_1, T_2, \ldots, T_{d-1}$. La racine de T_i est apellé l'**enfant** de r étiqueté par i ou le i-ème enfant de r.*

*Le **degré** (ou **arité**) d'un nœud est le nombre de ses enfants : les nœuds externes sont de degré 0. Le degré de l'arbre est le degré maximal de ses nœuds. Un arbre k-aire est un arbre ordonné où chaque nœud interne possède exactement k enfants. En particulier, un **arbre binaire**[6] est un arbre ordonné où chaque nœud interne possède exactement 2 enfants : les sous-arbres gauche et droit.*

[6] W(fr):arbr binaire

Représentation des nœuds de l'arbre

Arbre = ensemble d'objets représentant de nœuds + relations parent-enfant. En général, on veut retrouver facilement le parent et les enfants de n'importe quel nœud. Typiquement, les nœuds externes ne portent pas de données, et on les représente simplement par des liens null. Si l'arbre est d'arité k, on peut les stocker dans un tableau de taille k.

```
class TreeNode // noeud interne dans arbre binaire
{
   TreeNode parent; // null à la racine
   TreeNode left;   // enfant gauche, null=enfant externe
   TreeNode droit;  // enfant droit, null=enfant externe
   // ... d'autre information
}
class MultiNode // noeud interne de degré arbitraire
{
   MultiNode parent; // null à la racine
   MultiNode[] children; // enfants; children.length=arité
   // ... etc
}
```

First-child, next-sibling. Si l'arité de l'arbre n'est pas connu en avance, on peut utiliser une liste pour stocker les enfants : en mettant au nœud la tête de la liste de ses enfants, ainsi que la référence au prochain nœud sur sa propre liste des frères et sœurs, on obtient la représentation nommée **premier fils, prochain frère** (*first-child, next-sibling*), v. Fig. 8. (Le premier fils est la tête de la liste des enfants, et le prochain frère est le pointeur au prochain nœud sur la liste des enfants.) Dans cette représentation il faut stocker les nœuds externes explicitement (car le nœud externe peut avoir un nœud interne pour frère à la droite).

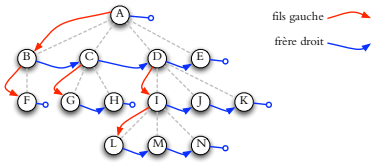

FIG. 8: Représentation premier-fils/prochain frère.

Théorème 4.1. *Il existe une correspondance 1-à-1 entre les arbres binaires à n nœuds internes et les arbres ordonnés à n nœuds (internes et externes).*

Démonstration. On peut intérprêter «premier fils» comme «enfant gauche», et «prochain frère» comme «enfant droit» pour obtenir un arbre binaire unique qui correspond à un arbre ordonné arbitraire, et vice versa. ■

Propriétés de nœuds

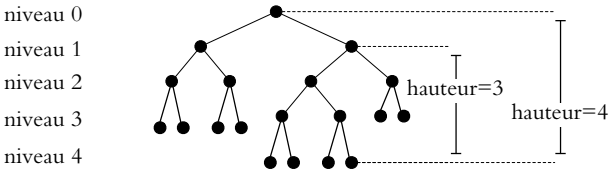

FIG. 9: Hauteur et niveau.

Niveau (*level/depth*) d'un nœud u : longueur du chemin qui mène à u à partir de la racine

Hauteur (*height*) d'un nœud u : longueur maximale d'un chemin de u jusqu'à un nœud externe dans le sous-arbre de u

Hauteur de l'arbre : hauteur de la racine (= niveau maximal de nœuds)

Longueur du chemin (interne/externe) (*internal/external path length*) somme des niveaux de tous les nœuds (internes/externes)

$$\mathsf{hauteur}[x] = \begin{cases} 0 & \text{si } x \text{ est externe ;} \\ 1 + \max_{y \in x.\mathsf{children}} \mathsf{hauteur}[y] \end{cases}$$

$$\mathsf{niveau}[x] = \begin{cases} 0 & \text{si } x \text{ est la racine } (x.\mathsf{parent} = \mathsf{null}) ; \\ 1 + \mathsf{niveau}[x.\mathsf{parent}] & \text{sinon} \end{cases}$$

Théorème 4.2. *Un arbre binaire à n nœuds externes contient $(n-1)$ nœuds internes.*

Un **arbre binaire complet** de hauteur h : il y a 2^i nœuds à chaque niveau $i = 0, \ldots, h-1$. On «remplit» les niveaux de gauche à droit.

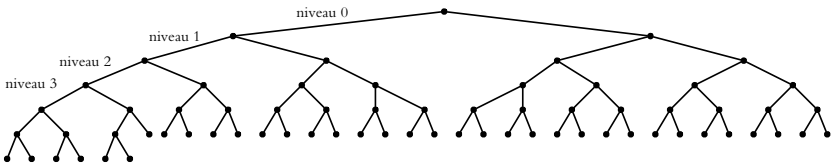

FIG. 10: Un arbre binaire complet.

Théorème 4.3. *La hauteur h d'un arbre binaire à n nœuds internes est bornée par* $\lceil \lg(n+1) \rceil \leq h \leq n$.

Démonstration. Un arbre de hauteur $h = 0$ ne contient qu'un seul nœud externe ($n = 0$), et les bornes sont correctes. Pour $h > 0$, on définit n_k comme le nombre de nœuds internes au niveau $k = 0, 1, 2, \ldots, h-1$ (il n'y a pas de nœud interne au niveau h). On a $n = \sum_{k=0}^{h-1} n_k$. Comme $n_k \geq 1$ pour tout $k = 0, \ldots, h-1$, on a que $n \geq \sum_{k=0}^{h-1} 1 = h$. Pour une borne supérieure, on utilise que $n_0 = 1$, et que $n_k \leq 2n_{k-1}$ pour tout $k > 0$. En conséquence, $n \leq \sum_{k=0}^{h-1} 2^k = 2^h - 1$, d'où $h \geq \lg(n+1)$. La preuve montre aussi les arbres extrêmes : une chaîne de nœuds pour $h = n$, et un arbre binaire complet. ■

Parcours

Un parcours visite tous les nœuds de l'arbre. Dans un **parcours préfixe** (*preorder traversal*), chaque nœud est visité avant que ses enfants soient visités. On calcule ainsi des propriétés avec récurrence vers le parent (comme niveau). Dans un **parcours postfixe** (*postorder traversal*), chaque nœud est visité après que ses enfants sont visités. On calcule ainsi des propriétés avec récurrence vers les enfants (comme hauteur).

Dans les algorithmes suivants, un nœud externe est null, et chaque nœud interne N possède les variables N.children (si arbre ordonné), ou N.left et N.right (si arbre binaire). L'arbre est stocké par une référence à sa racine root.

```
Algo PARCOURS-PRÉFIXE(x)
1  if x ≠ null then
2      «visiter» x
3      for y ∈ x.children do
4          PARCOURS-PRÉFIXE(y)
```

```
Algo PARCOURS-POSTFIXE(x)
1  if x ≠ null then
2      for y ∈ x.children do
3          PARCOURS-POSTFIXE(y)
4      «visiter» x
```

```
Algo NIVEAU(x, n)                    // remplit niveau[···]
// parent de x est à niveau n
// appel initiel avec x = root et n = −1
N1  if x ≠ null then
N2      niveau[x] ← n + 1            // (visite préfixe)
N3      for y ∈ x.children do
N4          NIVEAU(y, n + 1)
```

```
Algo HAUTEUR(x)                      // retourne hauteur de x
H1  max ← −1                         // (hauteur maximale des enfants)
H2  if x ≠ null then
H3      for y ∈ x.children do
H4          h ← HAUTEUR(y) ;
H5          if h > max then max ← h
H6  return 1 + max                   // (visite postfixe)
```

Lors d'un **parcours infixe** (*inorder traversal*), on visite chaque nœud après son enfant gauche mais avant son enfant droit. (Ce parcours ne se fait que sur un arbre binaire.)

```
Algo PARCOURS-INFIXE(x)
1 if x ≠ null then
2     PARCOURS-INFIXE(x.left)
3     «visiter» x
4     PARCOURS-INFIXE(x.right)
```

Un parcours préfixe peut se faire aussi à l'aide d'une pile. Si au lieu de la pile, on utilise une queue, alors on obtient un **parcours par niveau**.

```
Algo PARCOURS-PILE
1 initialiser la pile P
2 P.push(root)
3 while P ≠ ∅
4     x ← P.pop()
5     if x ≠ null then
6         «visiter» x
7         for y ∈ x.children do P.push(y)
```

```
Algo PARCOURS-NIVEAU
1 initialiser la queue Q
2 Q.enqueue(root)
3 while Q ≠ ∅
4     x ← Q.dequeue()
5     if x ≠ null then
6         «visiter» x
7         for y ∈ x.children do Q.enqueue(y)
```

Arbre syntaxique

Une expression arithmétique peut être representée par un **arbre syntaxique**. Parcours différents du même arbre mènent à des représentations différentes de la même expression.

Une opération arithmétique a op b est écrite en **notation polonaise inverse**[7] ou notation «postfixée» par $a\ b$ op. Avantage : pas de parenthèses ! Exemples : $1+2 \rightarrow 12+$, $(3-7)*8 \rightarrow 37-8*$. Une telle expression s'évalue à l'aide d'une pile : op $\leftarrow$ pop(), $b \leftarrow$ pop(), $a \leftarrow$ pop(), $c \leftarrow$ op(a,b), push(c). On répète le code tandis qu'il y a un opérateur en haut. À la fin, la pile ne contient que le résultat numérique. L'évaluation correspond à un parcours postfixe de l'arbre syntaxique.

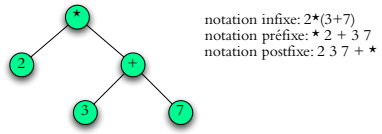

FIG. 11: Arbre syntaxique.L'arbre montre l'application de règles dans une grammaire formelle pour expressions : $E \rightarrow E+E|E*E|$*nombre*.

[7] W(fr):Notations infixée, préfixée, polonaise et postfixée

```
Algorithme EVAL(x)                    // (évalation de l'arbre syntaxique avec racine x)
E1 if x n'a pas d'enfants then return sa valeur // (une constante)
E2 else // (x est une opération op d'arité k)
E3     for i ← 0,...,k−1 do f_i ← EVAL(x.children[i])
E4     return le résultat de l'opération op avec les opérands (f_0, f_1, ..., f_{k−1})
```

La notation préfixe est généralement utilisée pour les appels de procédures, fonctions ou de méthodes dans des langages de programmation populaires comme Java et C. En même temps, les opérations arithmétiques et logiques sont typiquement écrites en notation infixe. Le langage **PostScript** utilise la notation postfixe partout, même en instructions de contrôle. En conséquence, l'interpréteur se relie sur des piles dans l'exécution. Le code `b {5 2 sub 20 moveto 30 40 lineto} if` dessine une ligne entre les points $(3,20)$ et $(30,40)$ si b est vrai. En Javaesque, on écrirait `if (b) {moveTo(5-2,20); lineTo(30, 40)}`. Le langage **Lisp** utilise la notation préfixe avec parènthèses obligatoires. En conséquence, les listes sont des objets fondamentaux ("Lisp"=*List Processing Language*) lors d'exécution : une liste est formée d'une tête (car pour Lispéens) et la liste de successeurs (cdr pour le Lispéen).

```
(with-canvas (:width 100 :height 200) ((moveto (- 5 2) 20) (lineto 30 40)))
```